1 *Boolean retrieval*

The meaning of the term *information retrieval* (IR) can be very broad. Just getting a credit card out of your wallet so that you can type in the card number is a form of information retrieval. However, as an academic field of study, *information retrieval* might be defined thus:

INFORMATION RETRIEVAL

> Information retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers).

As defined in this way, information retrieval used to be an activity that only a few people engaged in: reference librarians, paralegals, and similar professional searchers. Now the world has changed, and hundreds of millions of people engage in information retrieval every day when they use a web search engine or search their email.[1] Information retrieval is fast becoming the dominant form of information access, overtaking traditional database-style searching (the sort that is going on when a clerk says to you: "I'm sorry, I can only look up your order if you can give me your order ID").

Information retrieval can also cover other kinds of data and information problems beyond that specified in the core definition above. The term "unstructured data" refers to data that does not have clear, semantically overt, easy-for-a-computer structure. It is the opposite of structured data, the canonical example of which is a relational database, of the sort companies usually use to maintain product inventories and personnel records. In reality, almost no data are truly "unstructured." This is definitely true of all text data if you count the latent linguistic structure of human languages. But even accepting that the intended notion of structure is overt structure, most text has structure, such as headings, paragraphs, and footnotes, which is commonly represented in documents by explicit markup (such as the coding underlying web pages). Information retrieval is also used to facilitate "semistructured"

[1] In modern parlance, the word "search" has tended to replace "(information) retrieval"; the term "search" is quite ambiguous, but in context we use the two synonymously.

search such as finding a document where the title contains Java and the body contains threading.

The field of IR also covers supporting users in browsing or filtering document collections or further processing a set of retrieved documents. Given a set of documents, clustering is the task of coming up with a good grouping of the documents based on their contents. It is similar to arranging books on a bookshelf according to their topic. Given a set of topics, standing information needs, or other categories (such as suitability of texts for different age groups), classification is the task of deciding which class(es), if any, each of a set of documents belongs to. It is often approached by first manually classifying some documents and then hoping to be able to classify new documents automatically.

Information retrieval systems can also be distinguished by the scale at which they operate, and it is useful to distinguish three prominent scales. In *web search*, the system has to provide search over billions of documents stored on millions of computers. Distinctive issues are needing to gather documents for indexing, being able to build systems that work efficiently at this enormous scale, and handling particular aspects of the web, such as the exploitation of hypertext and not being fooled by site providers manipulating page content in an attempt to boost their search engine rankings, given the commercial importance of the web. We focus on all these issues in Chapters 19–21. At the other extreme is *personal information retrieval*. In the last few years, consumer operating systems have integrated information retrieval (such as Apple's Mac OS X Spotlight or Windows Vista's Instant Search). Email programs usually not only provide search but also text classification: they at least provide a spam (junk mail) filter, and commonly also provide either manual or automatic means for classifying mail so that it can be placed directly into particular folders. Distinctive issues here include handling the broad range of document types on a typical personal computer, and making the search system maintenance free and sufficiently lightweight in terms of startup, processing, and disk space usage that it can run on one machine without annoying its owner. In between is the space of *enterprise, institutional,* and *domain-specific search*, where retrieval might be provided for collections such as a corporation's internal documents, a database of patents, or research articles on biochemistry. In this case, the documents are typically stored on centralized file systems and one or a handful of dedicated machines provide search over the collection. This book contains techniques of value over this whole spectrum, but our coverage of some aspects of parallel and distributed search in web-scale search systems is comparatively light owing to the relatively small published literature on the details of such systems. However, outside of a handful of web search companies, a software developer is most likely to encounter the personal search and enterprise scenarios.

In this chapter, we begin with a very simple example of an IR problem, and introduce the idea of a term-document matrix (Section 1.1) and the

central inverted index data structure (Section 1.2). We then examine the Boolean retrieval model and how Boolean queries are processed (Sections 1.3 and 1.4).

1.1 An example information retrieval problem

A fat book that many people own is *Shakespeare's Collected Works*. Suppose you wanted to determine which plays of Shakespeare contain the words Brutus AND Caesar AND NOT Calpurnia. One way to do that is to start at the beginning and to read through all the text, noting for each play whether it contains Brutus and Caesar and excluding it from consideration if it contains Calpurnia. The simplest form of document retrieval is for a computer to do this sort of linear scan through documents. This process is commonly
GREP referred to as *grepping* through text, after the Unix command `grep`, which performs this process. Grepping through text can be a very effective process, especially given the speed of modern computers, and often allows useful possibilities for wildcard pattern matching through the use of regular expressions. With modern computers, for simple querying of modest collections (the size of *Shakespeare's Collected Works* is a bit under one million words of text in total), you really need nothing more.

But for many purposes, you do need more:

1. To process large document collections quickly. The amount of online data has grown at least as quickly as the speed of computers, and we would now like to be able to search collections that total in the order of billions to trillions of words.
2. To allow more flexible matching operations. For example, it is impractical to perform the query Romans NEAR countrymen with `grep`, where NEAR might be defined as "within 5 words" or "within the same sentence."
3. To allow ranked retrieval. In many cases, you want the best answer to an information need among many documents that contain certain words.

INDEX The way to avoid linearly scanning the texts for each query is to *index* the documents in advance. Let us stick with *Shakespeare's Collected Works*, and use it to introduce the basics of the Boolean retrieval model. Suppose we record for each document – here a play of Shakespeare's – whether it contains each word out of all the words Shakespeare used (Shakespeare used about 32,000
INCIDENCE different words). The result is a binary term-document *incidence matrix*, as in
MATRIX Figure 1.1. *Terms* are the indexed units (further discussed in Section 2.2); they
TERM are usually words, and for the moment you can think of them as words, but the information retrieval literature normally speaks of terms because some of them, such as perhaps I-9 or Hong Kong are not usually thought of as words. Now, depending on whether we look at the matrix rows or columns, we can

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth	...
Antony	1	1	0	0	0	1	
Brutus	1	1	0	1	0	0	
Caesar	1	1	0	1	1	1	
Calpurnia	0	1	0	0	0	0	
Cleopatra	1	0	0	0	0	0	
mercy	1	0	1	1	1	1	
worser	1	0	1	1	1	0	
...							

Figure 1.1 A term-document incidence matrix. Matrix element (t, d) is 1 if the play in column d contains the word in row t, and is 0 otherwise.

have a vector for each term, which shows the documents it appears in, or a vector for each document, showing the terms that occur in it.[2]

To answer the query Brutus AND Caesar AND NOT Calpurnia, we take the vectors for Brutus, Caesar and Calpurnia, complement the last, and then do a bitwise AND:

110100 AND 110111 AND 101111 = 100100

The answers for this query are thus *Antony and Cleopatra* and *Hamlet* (Figure 1.2).

BOOLEAN RETRIEVAL MODEL

The *Boolean retrieval model* is a model for information retrieval in which we can pose any query which is in the form of a Boolean expression of terms, that is, in which terms are combined with the operators AND, OR, and NOT. The model views each document as just a set of words.

DOCUMENT COLLECTION CORPUS

Let us now consider a more realistic scenario, simultaneously using the opportunity to introduce some terminology and notation. Suppose we have $N = 1$ million documents. By *documents* we mean whatever units we have decided to build a retrieval system over. They might be individual memos or chapters of a book (see Section 2.1.2 (page 20) for further discussion). We refer to the group of documents over which we perform retrieval as the (document) *collection*. It is sometimes also referred to as a *corpus* (a *body* of texts). Suppose each document is about 1,000 words long (2–3 book pages). If we assume an average of 6 bytes per word including spaces and punctuation, then this is a document collection about 6 gigabytes (GB) in size. Typically, there might be about $M = 500{,}000$ distinct terms in these documents. There is nothing special about the numbers we have chosen, and they might vary by an order of magnitude or more, but they give us some idea of the dimensions of the kinds of problems we need to handle. We will discuss and model these size assumptions in Section 5.1 (page 79).

AD HOC RETRIEVAL

Our goal is to develop a system to address the *ad hoc retrieval* task. This is the most standard IR task. In it, a system aims to provide documents from

[2] Formally, we take the transpose of the matrix to be able to get the terms as column vectors.

Antony and Cleopatra, Act III, Scene ii
Agrippa [Aside to Domitius Enobarbus]: Why, Enobarbus,
When Antony found Julius Caesar dead,
He cried almost to roaring; and he wept
When at Philippi he found Brutus slain.

Hamlet, Act III, Scene ii
Lord Polonius: I did enact Julius Caesar: I was killed i' the
Capitol; Brutus killed me.

Figure 1.2 Results from Shakespeare for the query Brutus AND Caesar AND NOT Calpurnia.

within the collection that are relevant to an arbitrary user information need, communicated to the system by means of a one-off, user-initiated query. An *information need* is the topic about which the user desires to know more, and is differentiated from a *query*, which is what the user conveys to the computer in an attempt to communicate the information need. A document is *relevant* if it is one that the user perceives as containing information of value with respect to their personal information need. Our example above was rather artificial in that the information need was defined in terms of particular words, whereas, usually a user is interested in a topic like "pipeline leaks" and would like to find relevant documents regardless of whether they precisely use those words or express the concept with other words such as pipeline rupture. To assess the *effectiveness* of an IR system (the quality of its search results), a user usually wants to know two key statistics about the system's returned results for a query:

INFORMATION NEED
QUERY
RELEVANCE
EFFECTIVENESS

PRECISION *Precision*: What fraction of the returned results are relevant to the information need?

RECALL *Recall*: What fraction of the relevant documents in the collection were returned by the system?

Detailed discussion of relevance and evaluation measures including precision and recall is found in Chapter 8.

We now cannot build a term-document matrix in a naive way. A 500K × 1M matrix has half-a-trillion 0's and 1's – too many to fit in a computer's memory. But the crucial observation is that the matrix is extremely sparse, that is, it has few nonzero entries. Because each document is 1,000 words long, the matrix has no more than one billion 1's, so a minimum of 99.8% of the cells are zero. A much better representation is to record only the things that do occur, that is, the 1 positions.

This idea is central to the first major concept in information retrieval, the *inverted index*. The name is actually redundant: an index always maps back from terms to the parts of a document where they occur. Nevertheless, *inverted index*, or sometimes *inverted file*, has become the standard term in

INVERTED INDEX

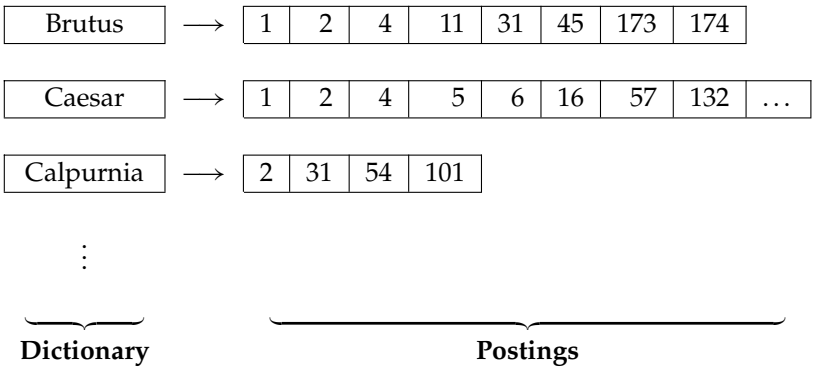

Figure 1.3 The two parts of an inverted index. The dictionary is commonly kept in memory, with pointers to each postings list, which is stored on disk.

IR.[3] The basic idea of an inverted index is shown in Figure 1.3. We keep a
DICTIONARY *dictionary* of terms (sometimes also referred to as a *vocabulary* or *lexicon*; in
VOCABULARY this book, we use *dictionary* for the data structure and *vocabulary* for the set of
LEXICON terms). Then, for each term, we have a list that records which documents the term occurs in. Each item in the list – which records that a term appeared in a document (and, later, often, the positions in the document) – is conventionally called a *posting*.[4] The list is then called a *postings list* (or inverted list), and all the postings lists taken together are referred to as the *postings*. The dictionary in Figure 1.3 has been sorted alphabetically and each postings list is sorted by document ID. We see why this is useful in Section 1.3; later, we also consider alternatives to doing this (Section 7.1.5).

POSTING
POSTINGS LIST
POSTINGS

1.2 A first take at building an inverted index

To gain the speed benefits of indexing at retrieval time, we have to build the index in advance. The major steps in this are:

1. Collect the documents to be indexed:

Friends, Romans, countrymen.	So let it be with Caesar	...

2. Tokenize the text, turning each document into a list of tokens:

Friends	Romans	countrymen	So	...

[3] Some IR researchers prefer the term inverted file, but expressions like index construction and index compression are much more common than inverted file construction and inverted file compression. For consistency, we use (inverted) index throughout this book.

[4] In a (nonpositional) inverted index, a posting is just a document ID, but it is inherently associated with a term, via the postings list it is placed on; sometimes we will also talk of a (term, docID) pair as a posting.

3. Do linguistic preprocessing, producing a list of normalized tokens, which are the indexing terms:

friend	roman	countryman	so	...

4. Index the documents that each term occurs in by creating an inverted index, consisting of a dictionary and postings.

We define and discuss the earlier stages of processing, that is, steps 1–3, in Section 2.2. Until then you can think of *tokens* and *normalized tokens* as also loosely equivalent to *words*. Here, we assume that the first three steps have already been done, and we examine building a basic inverted index by sort-based indexing.

Within a document collection, we assume that each document has a unique serial number, known as the document identifier (*docID*). During index construction, we can simply assign successive integers to each new document when it is first encountered. The input to indexing is a list of normalized tokens for each document, which we can equally think of as a list of pairs of term and docID, as in Figure 1.4. The core indexing step is *sorting* this list so that the terms are alphabetical, giving us the representation in the middle column of Figure 1.4. Multiple occurrences of the same term from the same document are then merged.[5] Instances of the same term are then grouped, and the result is split into a *dictionary* and *postings*, as shown in the right column of Figure 1.4. Because a term generally occurs in a number of documents, this data organization already reduces the storage requirements of the index. The dictionary also records some statistics, such as the number of documents which contain each term (the *document frequency*, which is here also the length of each postings list). This information is not vital for a basic Boolean search engine, but it allows us to improve the efficiency of the search engine at query time, and it is a statistic later used in many ranked retrieval models. The postings are secondarily sorted by docID. This provides the basis for efficient query processing. This inverted index structure is essentially without rival as the most efficient structure for supporting ad hoc text search.

DOCID

SORTING

DOCUMENT FREQUENCY

In the resulting index, we pay for storage of both the dictionary and the postings lists. The latter are much larger, but the dictionary is commonly kept in memory, and postings lists are normally kept on disk, so the size of each is important. In Chapter 5, we examine how each can be optimized for storage and access efficiency. What data structure should be used for a postings list? A fixed length array would be wasteful; some words occur in many documents, and others in very few. For an in-memory postings list, two good alternatives are singly linked lists or variable length arrays. *Singly linked lists* allow cheap insertion of documents into postings lists (following updates, such as when recrawling the web for updated documents), and naturally extend

[5] Unix users can note that these steps are similar to use of the `sort` and then `uniq` commands.

Doc 1
I did enact Julius Caesar: I was killed i' the Capitol; Brutus killed me.

Doc 2
So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious:

term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

⟹

term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

⟹

term	doc. freq.	→	postings lists
ambitious	1	→	2
be	1	→	2
brutus	2	→	1 → 2
capitol	1	→	1
caesar	2	→	1 → 2
did	1	→	1
enact	1	→	1
hath	1	→	2
I	1	→	1
i'	1	→	1
it	1	→	2
julius	1	→	1
killed	1	→	1
let	1	→	2
me	1	→	1
noble	1	→	2
so	1	→	2
the	2	→	1 → 2
told	1	→	2
you	1	→	2
was	2	→	1 → 2
with	1	→	2

Figure 1.4 Building an index by sorting and grouping. The sequence of terms in each document, tagged by their documentID (*left*) is sorted alphabetically (*middle*). Instances of the same term are then grouped by word and then by documentID. The terms and documentIDs are then separated out (*right*). The dictionary stores the terms, and has a pointer to the postings list for each term. It commonly also stores other summary information such as, here, the document frequency of each term. We use this information for improving query time efficiency and, later, for weighting in ranked retrieval models. Each postings list stores the list of documents in which a term occurs, and may store other information such as the term frequency (the frequency of each term in each document) or the position(s) of the term in each document.

to more advanced indexing strategies such as skip lists (Section 2.3), which require additional pointers. *Variable length arrays* win in space requirements by avoiding the overhead for pointers and in time requirements because their use of contiguous memory increases speed on modern processors with memory caches. Extra pointers can in practice be encoded into the lists as offsets. If updates are relatively infrequent, variable length arrays are more compact and faster to traverse. We can also use a hybrid scheme, with a linked list of fixed length arrays for each term. When postings lists are stored on disk, they are stored (perhaps compressed) as a contiguous run of postings without explicit pointers (as in Figure 1.3), so as to minimize the size of the postings list and the number of disk seeks to read a postings list into memory.

? **Exercise 1.1** [⋆] Draw the inverted index that would be built for the following document collection. (See Figure 1.3 for an example.)

Doc 1 new home sales top forecasts
Doc 2 home sales rise in july
Doc 3 increase in home sales in july
Doc 4 july new home sales rise

Exercise 1.2 [⋆] Consider these documents:

Doc 1 breakthrough drug for schizophrenia
Doc 2 new schizophrenia drug
Doc 3 new approach for treatment of schizophrenia
Doc 4 new hopes for schizophrenia patients

a. Draw the term-document incidence matrix for this document collection.

b. Draw the inverted index representation for this collection, as in Figure 1.3 (page 6).

Exercise 1.3 [⋆] For the document collection shown in Exercise 1.2, what are the returned results for these queries?

a. schizophrenia AND drug

b. for AND NOT (drug OR approach)

1.3 Processing Boolean queries

SIMPLE CONJUNCTIVE QUERIES

How do we process a query using an inverted index and the basic Boolean retrieval model? Consider processing the *simple conjunctive query*:

(1.1) Brutus AND Calpurnia

over the inverted index partially shown in Figure 1.3 (page 6). We:

1. Locate Brutus in the dictionary.
2. Retrieve its postings.
3. Locate Calpurnia in the dictionary.

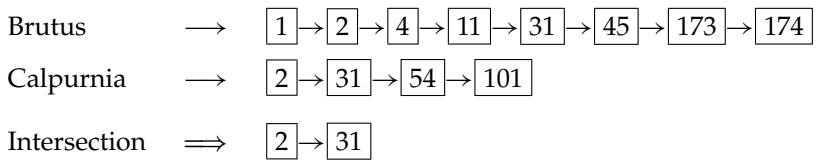

Figure 1.5 Intersecting the postings lists for Brutus and Calpurnia from Figure 1.3.

4. Retrieve its postings.
5. Intersect the two postings lists, as shown in Figure 1.5.

POSTINGS LIST
INTERSECTION
POSTINGS MERGE

The *intersection* operation is the crucial one: We need to efficiently intersect postings lists so as to be able to quickly find documents that contain both terms. (This operation is sometimes referred to as *merging* postings lists, this slightly counterintuitive name reflects using the term *merge algorithm* for a general family of algorithms that combine multiple sorted lists by interleaved advancing of pointers through each; here we are merging the lists with a logical AND operation.)

There is a simple and effective method of intersecting postings lists using the merge algorithm (see Figure 1.6): We maintain pointers into both lists and walk through the two postings lists simultaneously, in time linear in the total number of postings entries. At each step, we compare the docID pointed to by both pointers. If they are the same, we put that docID in the results list, and advance both pointers. Otherwise we advance the pointer pointing to the smaller docID. If the lengths of the postings lists are x and y, the intersection takes $O(x + y)$ operations. Formally, the complexity of querying is $\Theta(N)$, where N is the number of documents in the collection.[6] Our indexing methods gain us just a constant, not a difference in Θ time complexity compared with a linear scan, but in practice the constant is huge. To use this algorithm, it is crucial that postings be sorted by a single global ordering. Using a numeric sort by docID is one simple way to achieve this.

We can extend the intersection operation to process more complicated queries like:

(1.2) (Brutus OR Caesar) AND NOT Calpurnia

QUERY OPTIMIZATION

Query optimization is the process of selecting how to organize the work of answering a query so that the least total amount of work needs to be done by the system. A major element of this for Boolean queries is the order in which postings lists are accessed. What is the best order for query processing? Consider a query that is an AND of t terms, for instance:

(1.3) Brutus AND Caesar AND Calpurnia

[6] The notation $\Theta(\cdot)$ is used to express an asymptotically tight bound on the complexity of an algorithm. Informally, this is often written as $O(\cdot)$, but this notation really expresses an asymptotic upper bound, which need not be tight (Cormen et al. 1990).